Acta Univ. Sapientiae, Informatica, **2,** 2 (2010) 210–230

Modelling dynamic programming problems by generalized d-graphs

Zoltán Kátai
Sapientia Hungarian University of Transylvania
Department of Mathematics and Informatics,
Tg. Mureş, Romania
email: katai_zoltan@ms.sapientia.ro

Abstract. In this paper we introduce the concept of generalized d-graph (admitting cycles) as special dependency-graphs for modelling dynamic programming (DP) problems. We describe the d-graph versions of three famous single-source shortest algorithms (The algorithm based on the topological order of the vertices, Dijkstra algorithm and Bellman-Ford algorithm), which can be viewed as general DP strategies in the case of three different class of optimization problems. The new modelling method also makes possible to classify DP problems and the corresponding DP strategies in term of graph theory.

1 Introduction

Dynamic programming (DP) as optimization method was proposed by Richard Bellman in 1957 [1]. Since the first book in applied dynamic programming was published in 1962 [2] DP has become a current problem solving method in several fields of science: Applied mathematics [2], Computer science [3], Artificial Intelligence [6], Bioinformatics [4], Macroeconomics [13], etc. Even in the early book on DP [2] the authors drew attention to the fact that some dynamic programming strategies can be formulated as graph search problems. Later this subject was largely researched. As recent examples: Georgescu and Ionescu introduced the concept of DP-tree [7]; Kátai [8] proposed d-graphs as special hierarchic dependency-graphs for modelling DP problems; Lew and

Computing Classification System 1998: D.1
Mathematics Subject Classification 2010: 68W40, 90C39, 68R10, 05C12
Key words and phrases: dynamic programming, graph theory, shortest path algorithms

Mauch [14, 15, 16] used specialized Petri Net models to represent DP problems (Lew called his model Bellman-Net).

All the above mentioned modelling tools are based on cycle free graphs. As Mauch [16] states, circularity is undesirable if Petri Nets represent DP problem instances. On the other hand, however, there are DP problems with "cyclic functional equation" (the chain of recursive dependences of the functional equation is cyclic). Felzenszwalb and Zabih [5] in their survey entitled *Dynamic programming and graph algorithms in computer vision* recall that many dynamic programming algorithms can be viewed as solving a shortest path problem in a graph (see also [9, 11, 12]. But, interestingly, some shortest path algorithms work in cyclic graphs too. Kátai, after he has been analyzing the three most common single-source shortest path algorithms (The algorithm based on the topological order of the vertices, Dijkstra algorithm and Bellman-Ford algorithm), concludes that all these algorithms apply cousin DP strategies [10, 17]. Exploiting this observation Kátai and Csiki [12] developed general DP algorithms for discrete optimization problems that can be modelled by simple digraphs (see also [11]). In this paper, modelling finite discrete optimization problems by generalized d-graphs (admitting cycles), we extend the previously mentioned method for a more general class of DP problems. The presented new modelling method also makes possible to classify DP problems and the corresponding DP strategies in term of graph theory.

Then again the most common approach taken today for solving real-world DP problems is to start a specialized software development project for every problem in particular. There are several reasons why is benefiting to use the most specific DP algorithm possible to solve a certain optimization problem. For instance this approach commonly results in more efficient algorithms. But a number of researchers in the above mentioned various fields of applications are not experts in programming. Dynamic programming problem solving process can be divided into two steps: (1) the functional equation of the problem is established (a recursive formula that implements the principle of the optimality); (2) a computer program is elaborated that processes the recursive formula in a bottom-up way [12]. The first step is reachable for most researchers, but the second one not necessary. Attaching graph-based models to DP problems results in the following benefits:

- it moves DP problems to a well research area: graph theory,
- it makes possible to class DP strategies in terms of graph theory,
- as an intermediate representation of the problem (that hides, to some degree, the variety of DP problems) it enables to automate the program-

ming part of the problem-solving process by an adequately developed software-tools [12],

- a general software-tool that automatically solves DP problems (getting as input the functional equation) should be able to save considerable software development costs [16].

2 Modelling dynamic programming problems

DP can be used to solve optimization problems (discrete, finite space) that satisfy the principle of the optimality: *The optimal solution of the original problem is built on optimal sub-solutions respect to the corresponding sub-problems.* The principle of the optimality implicitly suggests that the problem can be decomposed into (or reduced to) similar sub-problems. Usually this operation can be performed in several ways. The goal is to build up the optimal solution of the original problem from the optimal solutions of its smaller sub-problems. Optimization problems can often be viewed as special version of more general problems that ask for all solutions, not only for the optimal one (A so-called objective function is defined on the set of sub-problems, which has to be optimized). We will call this general version of the problem, all-solutions-version.

The set of the sub-problems resulted from the decomposing process can adequately be modelled by dependency graphs (We have proposed to model the problem on the basis of the previously established functional equation that can be considered the output of the mathematical part and the input of the programming part of the problem solving process). The vertices (continuous/dashed line squares in the optimization/all-solutions version of the problem; see Figures 2.a,b,c) represent the sub-problems and directed arcs the dependencies among them. We introduce the following concepts:

- *Structural-dependencies:* We have directed arc from vertex A to vertex B if solutions of sub-problem A *may directly depend* on solutions of sub-problem B (dashed arcs; see Figure 2.a).
- *Optimal-dependencies:* We have directed arc from vertex A to vertex B if the optimal solution of sub-problem A *directly depends* on the optimal solution of the *smaller (respect to the optimization process)* sub-problem B (continuous arcs; see Figure 2.b).
- *Optimization-dependencies:* We have directed arc from vertex A to vertex B if the optimal solutions of sub-problem A *may directly depend* on

the optimal solution of the *smaller (respect to the optimization process)* sub-problem B (dotted arcs; see Figure 2.c).

Since structural dependencies reflect the structure of the problem, the *structural-dependencies-graph* can be considered as input information (It can be built up on the basis of the functional equation of the problem). This graph may contain cycles (see Figure 2.a). According to the principle of the optimality the *optimal-dependencies-graph* is a rooted sub-tree (acyclic sub-graph) of the structural-dependencies-graph. Representing the structure of the optimal solution the optimal-dependencies-graph can be viewed as output information. Since optimization-dependencies are such structural-dependencies that are *compatible* with the principle of the optimality, the *optimization-dependencies-graph* is a maximal rooted sub-tree of the structural-dependencies-graph that includes the optimal-dependencies-tree. Accordingly, the vertices of the optimization-dependencies-graph (and implicitly the vertices of the optimal-dependencies-graph too) can be arranged on levels (hierarchic structure) in such a way that all its arcs are directed downward. The original problem (or problem-set) is placed on the highest level and the trivial ones on the lowest level. We consider that a sub-problem is structurally trivial if cannot be decomposed into, or reduced to smaller sub-sub-problems. A sub-problem is considered to be trivial regarding the optimization process if its optimal solution trivially results from the input data. If the structural-dependencies-graph contains cycles, then completing the hierarchic optimization-dependencies-graph to the structural-dependencies-graph some added arcs will be directed upward.

Let us consider, as an example, the following problem: Given the weighted undirected triangle graph OAB determine

- all paths from vertex O (origin) to the all vertices (O, A, B) of the graph (Figure 1.a),
- the maximal length paths from vertex O (origin) to the all vertices of the graph ($|OA| = 10, |OB| = 10, |AB| = 100$) (Figure 1.b),
- the minimal length paths from vertex O (origin) to the all vertices of the graph ($|OA| = 100, |OB| = 10, |AB| = 10$) (Figure 1.c).

Since path (O,A,B) includes path (O,A) and, conversely, path (O,B,A) includes path (O,B) the structural-dependencies-graph that can be attached to the problem is not cycle free (Figure 2.a). We have bidirectional arcs between vertices representing sub-problems A (determine all paths to vertex A) and B (determine all paths to vertex B). Since the maximizing version of the problem does not satisfy the principle of the optimality (the maximal path (O,B,A)

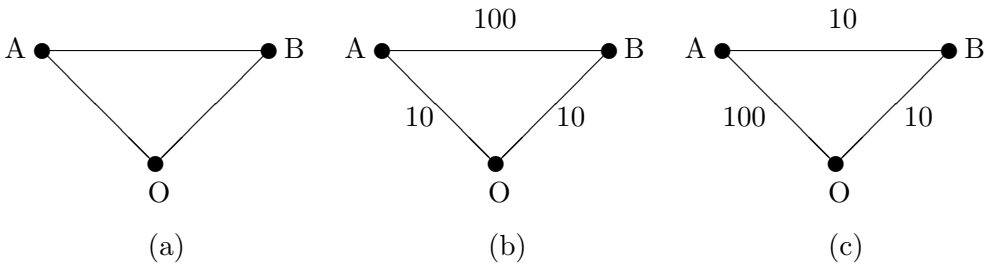


Figure 1: The triangle graph

includes path (O,B) that is not a maximal path too), in case b the optimal-dependencies-tree and the optimization-dependencies-tree are not defined. Figures 2.b and 2.c present the optimal- and optimization-dependencies-graphs attached to the minimizing version of the problem.

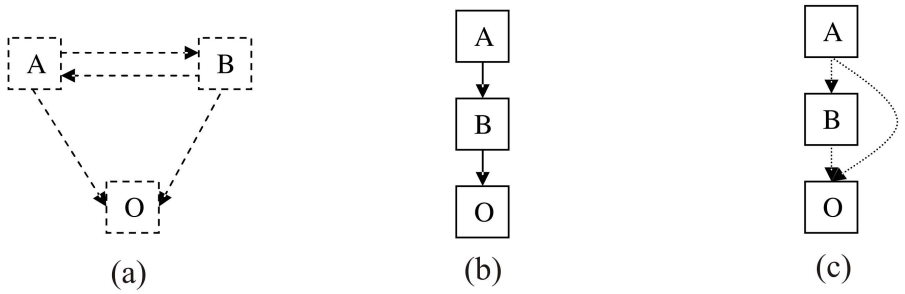


Figure 2: Structural/Optimal/Optimization-dependencies-graphs

3 d-graphs as special dependency graphs

Since decomposing usually means that the current problem is broken down into *two or more* immediate sub-problems ($1 \rightarrow N$ dependency) and since this operation can often be performed in *several* ways, Kátai [8] introduced d-graphs as special dependency graphs for modelling such problems. In this paper we define a generalized form of d-graphs as follows:

Definition 1 *The connected weighted bipartite finite digraph* $\mathsf{G_d(V,E,C)}$ *is a d-graph if:*

- $\mathsf{V = V_p \bigcup V_d}$ *and* $\mathsf{E = E_p \bigcup E_d}$, *where*
  - $\mathsf{V_p}$ *is the set of the p-vertices,*
  - $\mathsf{V_d}$ *is the set of the d-vertices,*
  - *all in/out neighbours of the p-vertices (excepting the source/sink vertices) are d-vertices; each d-vertex has exactly one p-in-neighbour; each d-vertex has at least one p-out-neighbour,*
  - $\mathsf{E_p}$ *is the set of p-arcs (from p-vertices to d-vertices),*
  - $\mathsf{E_d}$ *is the set of d-arcs (from d-vertices to p-vertices),*
- *function* $\mathsf{C : E_p \to \mathbb{R}}$ *associates a cost to every p-arc. We consider d-arcs of zero cost.*

If a d-graph is cycle-free, then its vertices can be arranged on levels (hierarchic structure) (see Figure 3). In [8] Kátai defines, respect to hierarchic d-graphs, the following related concepts: d-sub-graph, d-tree, d-sub-tree, d-spanning-tree, optimal d-spanning-tree and optimally weighted d-graph.

4 Modelling optimization problems by d-graphs

According to Kátai [8] a hierarchic d-graph can be viewed as representing the optimization-dependences-graph corresponding to the original problem and d-sub-graphs to the sub-problems. Since there is a one-to-one correspondence between p-vertices and d-sub-graph [8], these vertices also represent the sub-problems. The source p-vertex (or vertices) is attached to the original problem (or original problem-set), and the sink vertices to the structurally trivial sub-problems. A p-vertex has as many d-sons as the number of possibilities to decompose the corresponding sub-problem into its *smaller immediate* sub-sub-problems. A d-vertex has as many p-sons as the number of immediate smaller sub-problems (N) resulted through the corresponding *breaking-down* step ($1 \to \mathsf{N}$ dependency between the p-grandfather-problem and the corresponding p-grandson-problems). We will say that a grandfather-problem is *reduced to* its grandson-problem if the intermediary d-vertex has a single p-son ($1 \to 1$ dependency). Parallel decomposing processes may result in identical sub-problems, and, consequently, the corresponding p-vertex has multiple p-grandfathers (through different d-fathers). Due to this phenomenon the

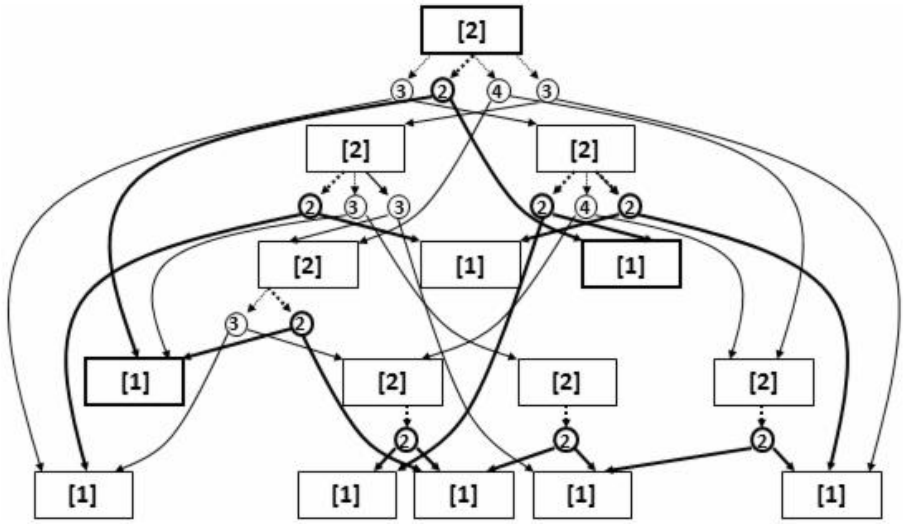


Figure 3: Hierarchic d-graph. p- and d-vertices are represented by rectangles and circles, respectively (We used bolded lines to emphasize the optimal d-spanning-(sub)trees)

number of the sub-problems may depend on the size of the input polynomially. The d-spanning-trees of the d-(sub)graphs represent the corresponding (sub)solutions, more exactly their tree-structure. The number of all solutions of the problem usually depends on the size of the input exponentially.

For example, if a p-vertex has n d-sons, these d-sons have $m_1, m_2, \ldots, m_n$ p-sons, and these p-son-problems have $(r_{1,1}, r_{1,2}, \ldots, r_{1,m1})$, $(r_{1,1}, r_{1,2}, \ldots, r_{1,m2})$, $(r_{1,1}, r_{1,2}, \ldots, r_{1,mn})$ solutions, respectively, then from the $\sum\sum r_{ij}$ solution of the p-grandson-problems results $\sum\prod r_{ij}$ solution for the common p-grandfather-problem. The number of solutions exponentially exceeds the number of sub-problems. The $\sum$-operator reflects the OR-connection between d-brothers and the $\prod$-operator the AND-connection between p-brothers.

5 Dynamic programming strategy on the optimization-dependencies d-graph

In the case of optimization problems we are interested only in the *optimal* solution of the original problem. Dynamic programming means building up

the optimal solutions of the *larger* sub-problems from the optimal solution of the already solved *smaller* sub-problems (starting with the optimal solution of the trivial sub-problems). Accordingly, (1) DP works on the hierarchic optimization-dependencies d-graph that can be attached to the problem, and (2) it deals with one solution per sub-problem, with the optimal one (DP strategies usually result in polynomial algorithms).

In line with this Kátai [8] defines two weight-functions ($w_p : V_p \to \mathbb{R}, w_d : V_d \to \mathbb{R}$) on the sets of p- and d-vertices of the attached hierarchic d-graph. Whereas the weight of a p-vertex is defined as the optimum (minimum/maximum) of the weights of its d-sons, the weight of a d-vertex is a function (depending on the problem to be modelled) of the weights of its p-sons. We consider the weight of a d-vertex to be optimal if is based on optimal the weights of its p-sons. The optimal weight of a p-vertex (excluding the sink vertices) is equal with the minimum/maximum of the optimal weights of its d-sons. The optimal weights of the p-sinks trivially result from the input data of the problem. Accordingly: the optimal weights of the p-vertices are computed (1) in *optimal way*, (2) on the basis of the *optimal weights* of their p-descendents. This means bottom-up strategy. Computing the optimal weights of the p-vertices we implicitly have their optimal d-sons (It is characteristic to DP algorithms that during the bottom-up building process it stores the already computed optimal p-weights in order to have them at hand in case they are needed to compute further optimal p-weights. If we also store the optimal d-sons of the p-vertices, then this information allows a quick reconstruction of the optimal d-spanning-tree in top-down way [17, 18]).

Defining the costs of the p-arcs as the absolute value of the weight-difference of its endpoints we get an optimally weighted d-graph with zero-cost minimal d-spanning-tree. We denote these kinds of p-arc-cost-functions by C^* [8]. Modelling optimization problems by a d-graphs $G_d(V, E, C^*)$ includes choosing functions w_p and w_d in such a way as the optimal weights of the p-vertices to represent the optimum values of the objective function respect to the corresponding sub-problems (These functions can be established on the basis of the functional equation of the problem; input information regarding the modelling process).

Proposition 2 *If an optimization problem can be modelled by a hierarchic d-graph* $G_d(V, E, C^*)$ *(as we described above), then it can be solved by dynamic programming.*

Proof. Since in an optimally weighted d-graph d-sub-trees of an optimal d-spanning-tree are also optimal d-spanning-trees respect to the d-sub-graphs

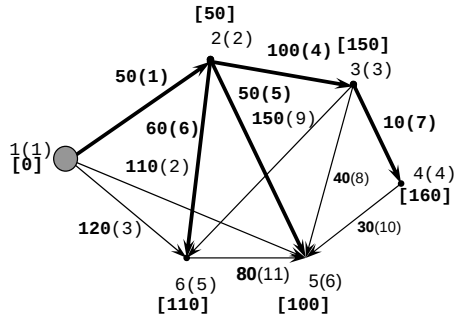
defined by their root-vertices, computing the optimal p- and d-weights according to a reverse topological order of the vertices (based on optimization-dependencies) implicitly identifies the optimal d-spanning-tree of the d-graph. This bottom-up strategy means DP and the optimal solution of the original problem will be represented by the weight of the source vertex (as value) and by the minimal d-spanning-tree (as structure). □

Computing the optimal weight of a p-vertex (expecting vertices representing trivial sub-problems) can be implemented as a gradual updating process based on the weights of its d-sons. The weights of p-vertices representing trivial sub-problems receive as starting-value their input optimal value. For other p-vertices we choose a proper starting-value according to the nature of the optimization problem (The weights of d-vertices are recomputed before every use). We define the following types of updating operations along p-arcs (if the weight of a certain d-son is "better" than the weight of his p-father, then the father's weight is replaced with the son's weight):

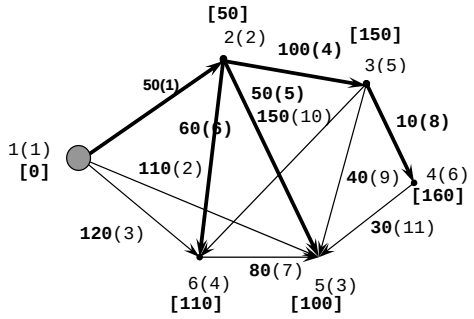
- *Complete:* based on the optimal value of the corresponding d-son.
- *Partial:* based on an intermediate value of the corresponding d-son.
- *Effective:* effectively improves the weight of the corresponding p-vertex.
- *Null:* dose not adjusts the weight of the corresponding p-vertex.
- *Optimal:* sets the optimal weight for the corresponding p-vertex. Optimal updates are complete and effective too.

6 d-graph versions of three famous single-source shortest-path algorithms

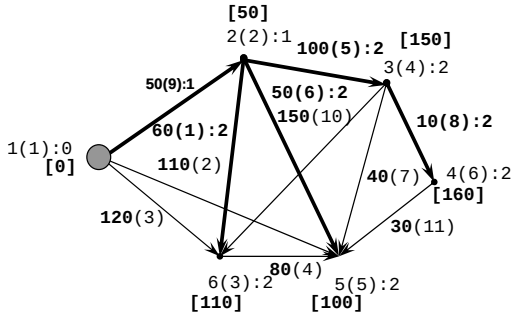
As we mentioned above, Kátai concludes that the three famous single-source shortest-path algorithms in digraphs (The algorithm based on the topological order of the vertices, Dijkstra algorithm and Bellman-Ford algorithm) apply cousin DP strategies [10, 17]. The common representative core of these DP algorithms is that the optimal weights (representing the optimal lengths to the corresponding vertices) are computed on account of updating these values along the arcs of the shortest-paths-tree according to their topological order (optimal-updating-sequence). Since this optimal tree is unknown (it represents the solution of the problem) all the three algorithms generate updating-sequences which contain, as subsequence, an optimal-updating-sequence necessary for the dynamic programming building process. The basic difference



(1,2), (1,5), (1,6), **(2,3)**, **(2,5)**, **(2,6)**, **(3,4)**, (3,5), (3,6), (4,5), (6,5)



(1,2), (1,5), (1,6), **(2,3)**, **(2,5)**, **(2,6)**, (6,5), **(3,4)**, (3,5), (3,6), (4,5)



1: (2,6), (1,5), (1,6), (6,5), (2,3), (2,5), (3,5), (3,4), **(1,2)**, (3,6), (4,5)

2: **(2,6)**, (1,5), (1,6), (6,5), **(2,3)**, **(2,5)**, (3,5), **(3,4)**, (1,2), (3,6), (4,5)

Figure 4: The strategies of the (a) Topological, (b) Dijkstra and (c) Bellman-Ford algorithms (we bolded the optimal-arc-subsequence of the generated arc-sequences)

among the three algorithms is the way they generate a proper arc-sequence and the corresponding updating-sequence.

In case the input digraph is acyclic we get a proper arc-sequence by ordering all the arcs of the graph topologically (this order can even be determined in advance). Dijkstra algorithm (working in cyclic graphs too, but without negative weighted arcs) determines the needed arc-sequence on the fly (parallel with the updating process). After the current weight of the next closest vertex has been confirmed as optimal value (greedy decision), the algorithm performs updating operations along the out-arcs of this vertex (This greedy choice can be justified as follows: if other out-neighbours of the growing shortest-paths-tree are farther - from the source vertex - than the currently closest one, then through these vertices cannot lead shortest paths to this). Bellman-Ford algorithm (working in cyclic graphs with negative weighted arcs too, but without feasible negative weighted cycles) goes through (in arbitrary order) all the arcs of the graph again and again until the arc-sequence generated in this way finally will contains, as sub-sequence, an optimal-updating-sequence (see Figure 4, [10]). The following d-graph algorithms implement DP strategies that exploit the core idea behind the above described single-source shortest-paths algorithms.

6.1 Building-up the optimization-dependencies d-graph in bottom-up way

Our basic goal is to perform updating operation along the p-arcs of the optimal-dependencies-tree according to their reverse topological order. We will call such arc sequences optimal-arc-sequence and the corresponding updating sequences optimal-updating-sequence. An optimal-updating-sequence surely results in building up the optimal value representing the optimal solution of the problem. Since the optimal-dependencies-tree is unknown (it represents the structure of the optimal solution to be determined), we should try to elaborate *complete arc sequences* that includes the desired optimal-updating-sequence (gratuitous updating operations have, at the worst, null effects).

We introduce the following colouring-convention:

- Initially all vertices are white.
- A p-vertex changes its colour to grey after the first attempt to update its weight. d-vertices automatically change their colour to grey if they do not have any more white p-sons.
- When the weight of a vertex riches its optimal value its colour is automatically changed to black.

We are facing a gratuitous updating operation if:

- along the corresponding p-arc was previously performed a complete update,
- the corresponding p-father is already black,
- the corresponding d-son is still grey or white.

Since the optimal values of trivial sub-problems automatically results from the input data of the problem, the corresponding p-vertices are automatically coloured with black.

The following propositions can be viewed as theoretical support for the below strategies that build up the optimal-dependencies d-graph (on the basis of the structural-dependencies-graph that can be considered input information) level-by-level in bottom-up way (At the beginning all p-vertices are places at level 0. All effective updates along the corresponding p-arcs move their p-end to higher level than the highest p-son of their d-end.).

Proposition 3 *If the structural-dependencies d-graph attached to an optimization problem that satisfies the principle of the optimality has no black p-sources, then there exists at least one p-arc corresponding to an* effective complete *updating operation.*

Proof. Since the optimization problem satisfies the principle of the optimality the optimal-updating-sequence there exists and continuously warrants (while no black p-sources still exist) the existence of optimal updating operations, which are effective and complete too. □

Proposition 4 *Any p-arcs sequence (of the structural-dependencies d-graph attached to an optimization problem that satisfies the principle of the optimality) that continuously applies non-repetitive complete updates (while such updating operations still exist) warrants that all p-sources become black-coloured. These p-arcs sequences contain arcs representing optimization-dependencies and surely include an optimal-arc-sequence.*

Proof. Since the optimization problem satisfies the principle of the optimality the optimal-updating-sequence there exists and warrants the continuous existence of optimal updating operations (which are also effective and complete) while no black p-sources still exist. Accordingly any p-arcs sequence that continuously applies non-repetitive complete updates includes an optimal-arc-sequence, and consequently results in colouring all p-sources with black. □

Proposition 5 *If the structural-dependencies d-graph attached to an optimization problem that satisfies the principle of the optimality is cycle-free, then any reverse topological order of the all p-arcs continuously applies non-repetitive complete updates, and consequently, results in building up the optimal solution of the problem.*

Proof. Since the colours of the d-vertices surely become black after all their p-sons have already become black, any reverse topological order of all p-arcs continuously applies non-repetitive complete updates. According to the previous proposition these arc-sequences surely include an optimal-arc-sequence, and consequently results in building up the optimal solution of the problem. □

Proposition 6 *If an optimization problem satisfies the principle of the optimality, then there exists a finite multiple complete arc-sequence of the attached structural-dependencies d-graph that includes an optimal-arc-sequence, and consequently, the corresponding updating-sequence results in building up the optimal solution of the problem.*

Proof. The existence of such an arc-sequence immediately results from the facts that: (1) Any complete arc-sequence contains all arcs of the optimal-dependencies-tree; (2) The optimal-dependencies-tree is finite. If we repeat a complete arc-sequence that includes the arcs of the optimal-dependencies-tree according to their topological order (worst case), then we need as many updating-tours as the number of the p-arcs of the optimal-dependencies-tree is. □

6.1.1 Algorithm d-TOPOLOGICAL

If the structural-dependencies d-graph attached to the problem is cycle free (called: structurally acyclic DP problems), then this input graph can also be viewed as optimization-dependencies-graph. Considering a reverse topological order of the *all* vertices, all updating operations (along the corresponding p-arc-sequence) will be complete (see Proposition 5). Additionally, along the arcs of the optimal d-spanning-tree we have optimal updates. Accordingly, this algorithm (called d-TOPOLOGICAL) results in determining the optimal solution of the problem.

6.1.2 Algorithm d-DIJKSTRA

If the structural-dependencies d-graph contains cycles a proper vertices order involving complete updates along the corresponding p-arc-sequence cannot be structurally established. In this case we should try to build up the

optimization-dependencies d-graph (more exactly a reverse topological order of its all p-arcs) on the fly, parallel with the bottom-up optimization process.

Implementing a sequence of continuous complete updates presumes to identify at each stage of the building process the black d-vertices based on which we have not performed complete updating operations (Proposition 3 guaranties that such d-vertices exist continuously). A d-vertex is black only if all its p-sons are already black. Consequently, the basic question is: Can we identify the black p-vertices at each stage of the building process? As we mentioned above a p-vertex is certainly black after we have performed complete updates based on *all* its d-sons (The last effective update was performed on the basis of optimal d-son). Algorithms based on the topological order of the all arcs exploit this *structural* condition. However, a p-vertex may have become black before we have performed complete updating operation along all its p-out-arcs. Conditions making perceptible such black p-vertices may also be deduced from the principle of the optimality. For example, if the DP problem has a greedy character too, then it may work the following condition: the "best" d-vertex (having *relatively* optimal weight) among those based on which we have not performed complete updating operations can be considered black (Called: Cyclic DP problems characterized by greedy choices). Since Dijkstra algorithm applies this strategy, we call this algorithm: d-DIJKSTRA.

6.1.3 Algorithm d-BELLMAN-FORD

If we cannot establish one complete arc-sequence including an optimal-arc-sequence (we will call such problems: DP problems without 'negative cycles'), we are forced to repeat the updating-tour along a complete (even arbitrary) arc-sequence of the input graph (structural-dependencies d-graph) until this *multiple arc-sequence* will include the desired optimal updating sequence (see Proposition 6). An extra tour without any effective updates indicates that the optimal solution has been built up. If the arbitrary arc-sequence we have chosen includes the arcs of the optimal-dependencies-tree in topological order (worst case), then we need as many updating-tours as the number of the p-arcs of the optimal-dependencies-tree is. Since Bellman-Ford algorithm applies this strategy, we call this algorithm: d-BELLMAN-FORD.

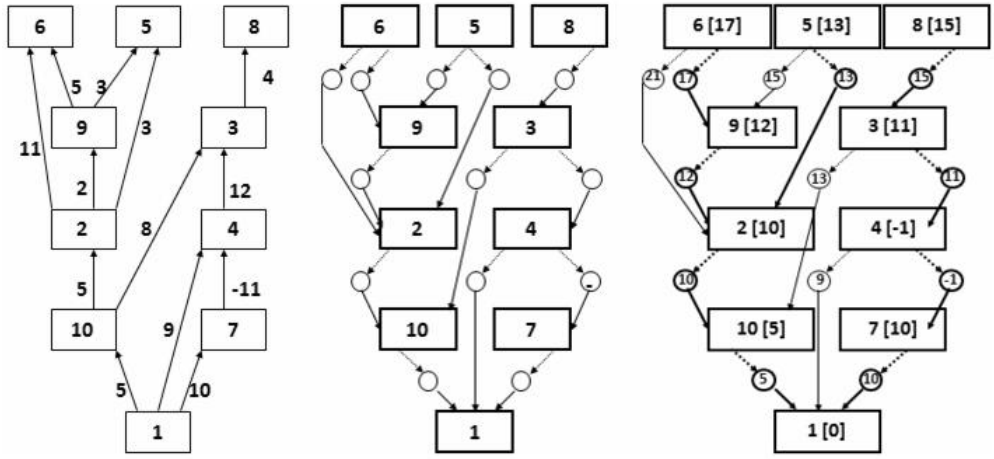
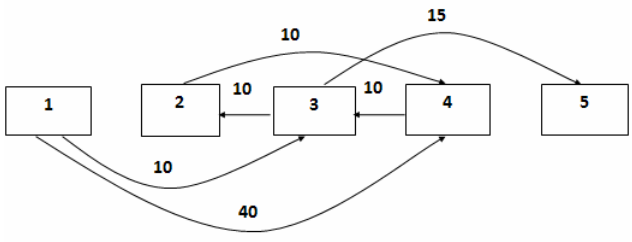
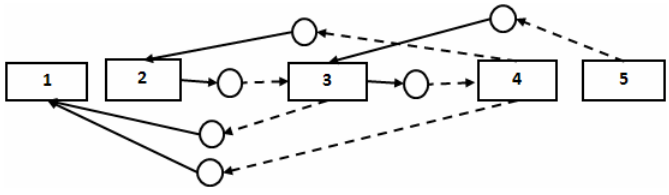


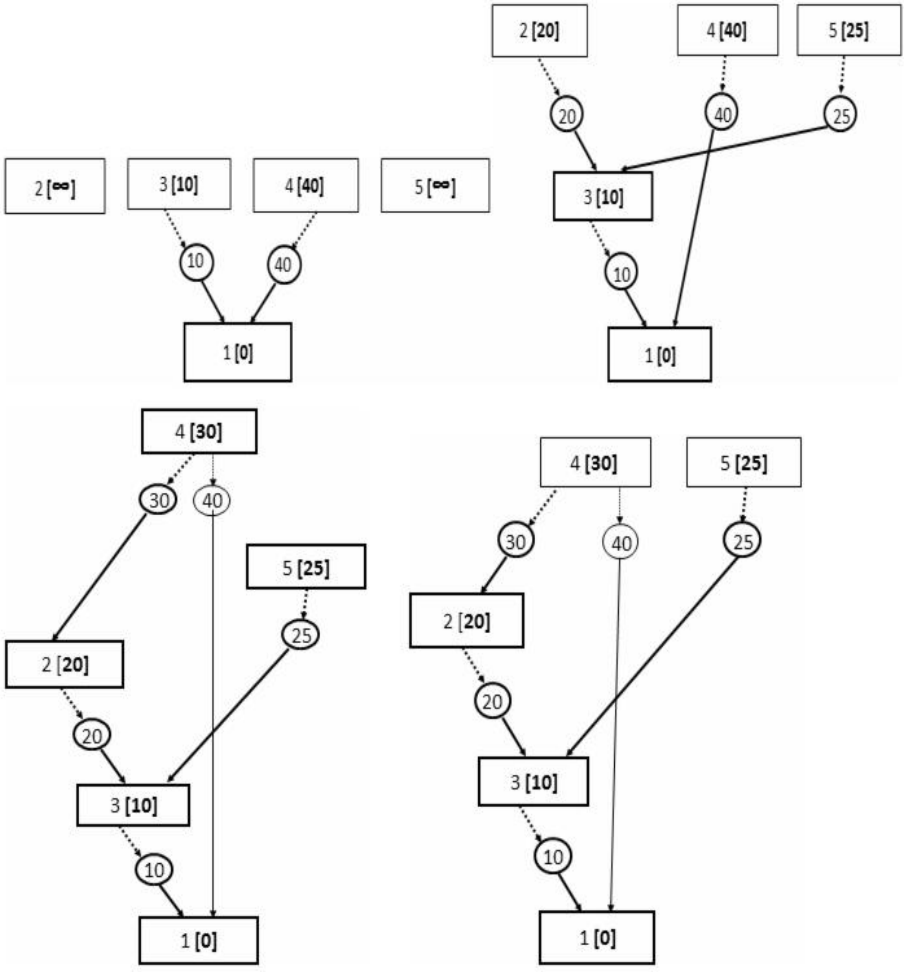
Figure 5: (a) Acyclic digraph; (b) Structural-dependencies d-graph; (c) Optimally weighted optimization-dependencies d-graph (bolded lines represent the arcs of the optimal-dependencies d-graph)



(a)

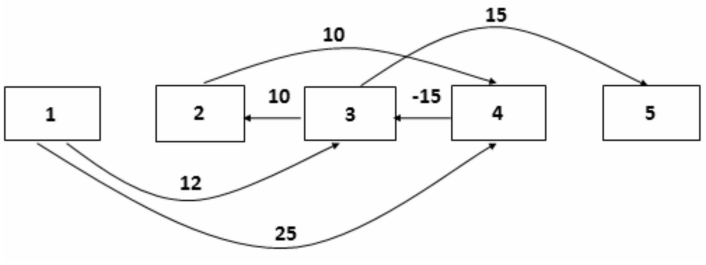


(b)

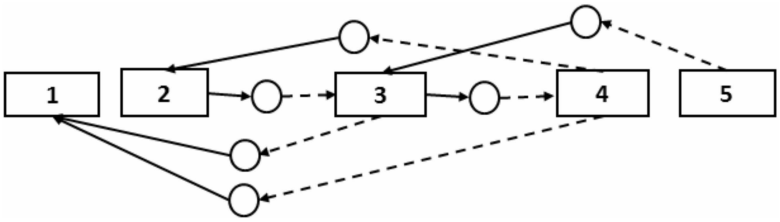


(c)

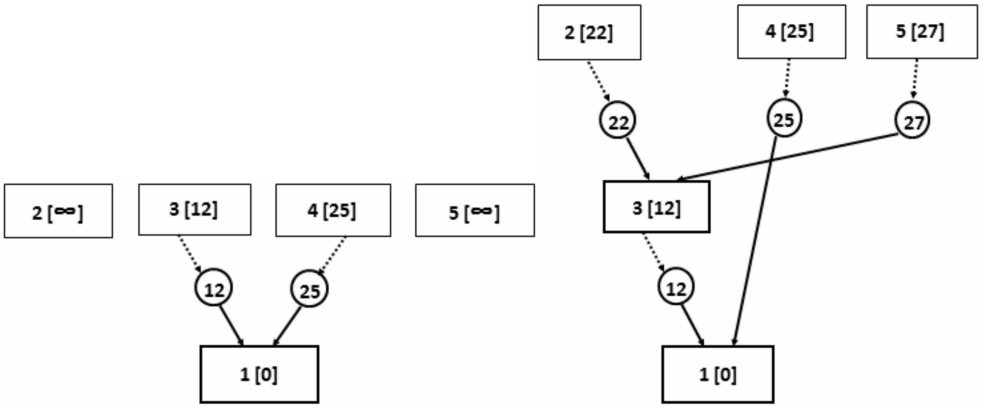
Figure 6: (a) Cyclic digraph without negative weighted arcs; (b) Cyclic structural-dependencies d-graph; (c) The bottom-up building-up process of the optimally weighted optimization-dependencies d-graph (bolded lines represent the arcs of the optimal-dependencies d-graph)


15
10
1
2
10
3
-15
4
5
12
25

(a)


1
2
3
4
5

(b)


2 [∞]
3 [12]
4 [25]
5 [∞]
12
25
1 [0]
2 [22]
4 [25]
5 [27]
22
25
27
3 [12]
12
1 [0]

(c1) (c2)

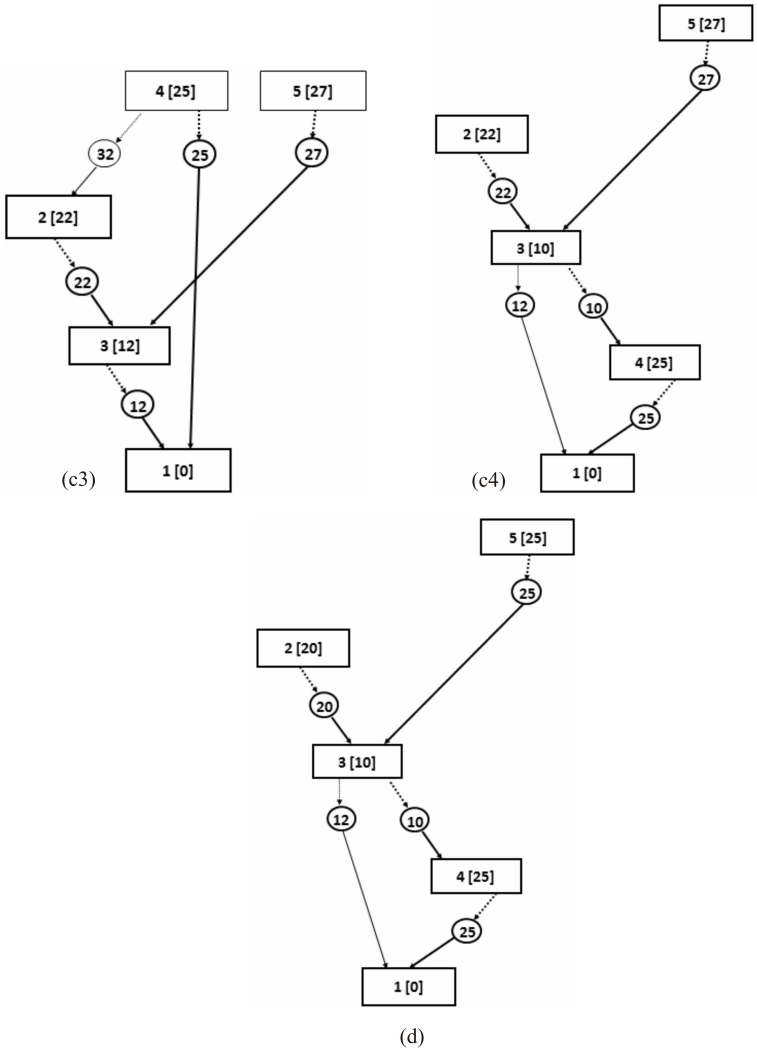


Figure 7: (a) Cyclic digraph with negative weighted arcs, but without negative cycles; (b) Cyclic structural-dependencies d-graph; The bottom-up building-up process of the optimally weighted optimization-dependencies d-graph (bolded lines represent the arcs of the optimal-dependencies d-graph): (c1–c4) first updating-tour, (d) second updating-tour

6.1.4 A relevant sample problem

As an example we consider the single-source shortest problem: Given a weighted digraph determine the shortest paths from a source vertex to all the other vertices (destination vertices). The attached figures (see Figures 5, 6, 7) illustrate the level by level building process of the optimization-dependencies d-graph concerning to the algorithms d-TOPOLOGICAL, d-DIJKSTRA and d-BELLMAN-FORD (Regarding this problem we have only $1 \to 1$ dependencies between neighbour p-vertices).

7 Conclusions

Introducing the generalized version of d-graphs we received a more effective tool for modelling a larger class of DP problems (Hierarchic d-graphs introduced in [8] and Petri-net based models [14, 15, 16] work only in the case of structurally acyclic problems; Classic digraphs [11, 12] can be applied when during the decomposing process at each step the current problem is reduced to only one sub-problem). The new modelling method also makes possible to classify DP problems (Structurally acyclic DP problems; Cyclic DP problems characterized by greedy choices; DP problems without 'negative cycles') and the corresponding DP strategies (d-TOPOLOGICAL, d-DIJKSTRA, d-BELLMAN-FORD) in term of graph theory.

If we have proposed to develop a general software-tool that automatically solves DP problems (getting as input the functional equation) we should combine the above algorithms as follows:

- We represent explicitly the d-graph described implicitly by the functional equation.
- We try to establish the reverse topological order of the vertices by a DFS like algorithm (d-DFS). This algorithm can also detect possible cycles.
- If the graph is cycle free, we apply algorithm d-TOPOLOGICAL, else we try to apply algorithm d-DIJKSTRA.
- If no mathematical guarantees that we reached the optimal solution, then choosing as complete arc-sequence for algorithm d-BELLMAN-FORD the arc-sequence generated by algorithm d-DIJKSTRA (completed with unused arcs) in the first updating-tour we verify the d-DIJKSTRA result. We repeat the updating tours until no more effective updates.

Such a software-application should be able to save considerable software development costs.

8 Acknowledgements

This research was supported by the Research Programs Institute of Sapientia Foundation, Cluj, Romania.

References

[1] R. Bellman, *Dynamic programming*, Princeton University Press, Princeton, NJ, 1957. ⇒210

[2] R. Bellman, S. Dreyfus, *Applied dynamic programming*, Princeton University Press, Princeton, NJ, 1962. ⇒210

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to algorithms,* 3rd edition, The MIT Press, Cambridge, MA, USA, 2009. ⇒210

[4] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, *Biological sequence analysis,* Cambridge University Press, Cambridge, UK, 1998. ⇒210

[5] P. F. Felzenszwalb, R. Zabih, Dynamic programming and graph algorithms in computer vision, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Preprint, 19 July 2010, 51 p. http://doi.ieeecomputersociety.org/10.1109/TPAMI.2010.135 ⇒211

[6] Z. Feng, R. Dearden, N Meuleau, R. Washington, Dynamic programming for structured continuous Markov decision problems, *Proc. 20th Conference on Uncertainty in Artificial Intelligence, ACM International Conference Proceeding Series*, AUAI Press, **70** (2004) pp. 154–161. ⇒210

[7] H. Georgescu, C. Ionescu, The dynamic programming method, a new approach, *Studia Univ. Babeş-Bolyai Inform.*, **43,** 1 (1998) 23–38. ⇒210

[8] Z. Kátai, Dynamic programming and d-graphs, *Studia Univ. Babeş-Bolyai Inform.*, **51,** 2 (2006) 41–52. ⇒210, 214, 215, 217, 228

[9] Z. Kátai, Dynamic programming strategies on the decision tree hidden behind the optimizing problems, *Informatics in Education*, **6,** 1 (2007) 115–138. ⇒211

[10] Z. Kátai, The single-source shortest paths algorithms and the dynamic programming, *Teaching Mathematics and Computer Science*, **6,** Special Issue (2008) 25–35. ⇒211, 218, 220

[11] Z. Kátai, Dynamic programming as optimal path problem in weighted digraphs, *Acta Math. Acad. Paedagog. Nyházi*, **24,** 2 (2008) 201–208. ⇒ 211, 228

[12] Z. Kátai, A. Csiki, Automated dynamic programming, *Acta Univ. Sapientiae Inform.*, **1,** 2 (2009) 149–164. ⇒211, 212, 228

[13] I. King, *A simple introduction to dynamic programming in macroeconomic models*, 2002.
http://researchspace.auckland.ac.nz/bitstream/handle/2292/190/230.pdf
⇒210

[14] A. Lew, A Petri net model for discrete dynamic programming, *Proc. 9th Bellman Continuum: International Workshop on Uncertain Systems and Soft Computing,* Beijing, China, July 24–27, 2002, pp. 16–21. ⇒211, 228

[15] A. Lew, H. Mauch, Bellman nets: A Petri net model and tool for dynamic programming, *Proc. 5th Int. Conf. Modelling, Computation and Optimization in Information Systems and Management Sciences* (MCO), Metz, France, 2004, pp. 241–248. ⇒211, 228

[16] H. Mauch, DP2PN2Solver: A flexible dynamic programming solver software tool, *Control Cybernet.*, **35**, 3 (2006) 687–702. ⇒211, 212, 228

[17] M. Sniedovich, Dijkstra's algorithm revisited: the dynamic programming connexion, *Control Cybernet.*, **35,** 3 (2006) 599–620. ⇒211, 217, 218

[18] D. Vagner, Power programming: dynamic programming, *The Mathematica Journal*, **5,** 4 (1995) 42–51. ⇒217

Received: September 1, 2010 • Revised: November 10, 2010